Robot Learning

Coursework 2

Released on Wednesday 7th February 2024

Submission deadline: **Friday 1st March at 7pm**

Written by Edward Johns

Please download the "Data File" available on Scientia, which accompanies this document.

Introduction.

Welcome to Coursework 2!

In Coursework 2, you will design and implement your own robot learning method, which you will write in *robot.py*. Then, we will evaluate your method, by testing your *robot.py* on random environments using our own PC, and scoring your method based on how well the robot performs. There are no more lab exercises being released for this module, so you are free to work on this coursework during the remaining lab sessions. This coursework will help you to better understand how to implement a variety of methods you have been learning about, including model-based reinforcement learning, imitation learning, and hybrid methods that combine imitation learning and model-based reinforcement learning.

Submission.

Your code should be submitted on LabTS, where you should provide your *robot.py* file (and optionally your *configuration.py* file if required by your *robot.py* file). You should also submit a single page PDF titled *coursework_2.pdf* on Scientia, where you will describe your implementation and experiments. Both *robot.py* and *coursework_2.pdf* will count towards your coursework grade. Your grade for *robot.py* will contribute to 80% of your grade for this coursework, and your grade for *robot-learning-coursework-2-description.pdf* will contribute to 20% of your grade for this coursework. And this coursework contributes to 15% of the entire module.

Setting Up.

There are the following six Python files provided in "Data File": *configuration.py*, *constants.py*, *environment.py*, *graphics.py*, *robot.py*, and *robot-learning.py*. You should be familiar with each of these from the tutorials. To begin, create a Python virtual environment with Python 3.10, and install the following libraries: *numpy* (version 1.26.4), *torch* (version 2.2.0), *pyglet* (version 2.0.10), *scipy* (version 1.12.0), *perlin-noise* (version 1.12), and *matplotlib* (version 3.8.2). Note that it is very important that you use Python 3.10 and install

these exact library version numbers, so that your *robot.py* will run without error when we evaluate your code.

The Window

Left Side of the Window.

Try running *robot-learning.py*. On the left of the window, is the live environment, as with the tutorials. The robot is the blue circle, and the task is for the robot to reach the goal, which is the green circle. In the tutorials, the background was a uniform colour, but now the background is a random terrain showing dark regions and light regions. This represents part of the environment dynamics: the robot can move faster through the dark regions, but only slowly through the light regions. You can image this to be like "mountains" and "valleys", where the robot can move faster through the valleys, but it is slowed down if it must climb over a mountain. Currently, the robot executes random actions, and you should see the effect of the environment dynamics as the robot moves quickly or slowly over different regions. Whenever the robot starts a new episode, either during training or testing, it begins randomly inside the initial region, which is the light blue square.

Middle of the Window.

The middle of the window is some free space where you can visualise whatever paths you like, to help you develop your method or to debug. If you inspect the *Robot* class in *robot.py*, you will see that it has an attribute called *Robot.paths_to_draw*. When you add a path to this list, it will automatically be drawn in the middle of the window. Each path should be a *PathToDraw* object, and this class is defined in *graphics.py*. You can see in this class that you can define not only the path (a list of states), but also the colour and thickness of the line. Therefore, you can be creative in the way you visualise whatever paths you want, such as the paths that have been planned, the demonstration paths that have been acquired, or any other paths you would like to visualise.

Right of the Window.

As with Tutorial 3, the right of the window visualises both the true environment dynamics, and the robot's model of this dynamics. This will be very helpful in understanding how well your model has been learned if you implement a model-based reinforcement learning method.

The Task

Training and Testing.

Your task is to develop a robot learning method in *robot.py*, which will enable the robot to learn to reach the goal. When we evaluate your code, we will first train the robot for a period of time, and then test the robot to see if it can reach the goal. You will be given a score based on how quickly the robot can reach the goal during testing, or how close it could get to the goal if it could not reach it.

The Operations Available.

Essentially, there are four operations that the robot can do during training: (1) take a step in the environment, (2) do some data processing / network training, (3) request and process a

demonstration, (4) request an environment reset. When a demonstration is requested, a list of states and actions is provided to the robot, which you can use to implement an imitation learning strategy. However, these demonstrations are not perfect, and each one is different. When an environment reset is requested, the robot will be moved to a random state in the initial region.

The Budget.

To make this task a little more realistic, the amount of training that the robot is allowed to do is modelled on a "budget of money" which you have, and there are various different costs associated with the different operations available. Each of the above four operations is assigned a "cost", the values of which can be seen in *constants.py*. So, each time the robot takes a step in the environment, or each time the robot takes some time to do some planning or some training, or each time a demonstration is requested, or each time an environment reset is requested, this costs money, and the overall money remaining is reduced. Once there is no money remaining, the training period is over, and the testing begins.

Requesting a demonstration is the most expensive, and resetting the environment is also quite expensive, but it is less laborious than providing a demonstration, and so it is cheaper. The cheapest operation is for the robot to take a step in the environment. A little more expensive than that is the cost per second of CPU time, which will accumulate when you train any networks or do any computationally expensive planning.

The Starter Code

robot-learning.py

If you inspect *robot-learning.py*. You will see that there are some similarities between the code in the tutorials, but some differences too. One difference is that there is now a "training" and "testing" mode. During the training mode, the robot will explore the environment and learn how to reach the goal. During the testing model, the robot will try to reach the goal as quickly as possible, without doing any further training.

As with the tutorials, the *update()* function is repeatedly called, and each time, the code receives a request from the robot for one of the above four operations, and then processes that request. The function *Robot.get_next_action_type()* must return either "demo", "reset", or "step", and then in the main loop in *robot-learning.py*, this request is processed accordingly. Take some time to inspect *robot-learning.py* further and try to understand the main program flow.

robot.py

This is where all your implementation work will go. You may wish to modify the other files for debugging, but when you test your method before submitting your code on Scientia, be sure to test it with the original files that were provided for the coursework. If you have made any modifications to the other files that your method depends on, remember that they are not going to be available when we evaluate your code, because you will only be submitting *robot.py*.

Initially, the *Robot* class has a minimum set of functions, which enable *robot-learning.py* to request operations from the robot, and then send back relevant information. You will need to modify the contents of these functions, but you should not change their names of their arguments because they are called by *robot-learning.py*. For example, in

Robot.get_next_action_testing(), you will need to return the action that the robot should execute during testing. And this could be the prediction from a policy network, or an action calculated from planning, for example. You will probably also need to create new functions in order to put together a full robot learning method.

environment.py

As explained above, one component of the environment dynamics is the speed at which the robot can move through different regions of the environment. But there is also another component of the dynamics, which is the direction in which the robot moves on each time step, for a given action. There is a non-linear mapping between the action and this direction, and this non-linear mapping is also a function of the robot's state, which is randomly set for each environment. You may wish to inspect *Environment.dynamics()* to try to understand how the dynamics is determined. However, it is not essential to understand this; remember that one of the benefits of model-based reinforcement learning is that the model can be learned without having any prior knowledge of the true dynamics.

What to Implement for robot.py

Your Own Robot Learning Method!

You are free to implement whatever robot learning method you like, taking inspiration from the lectures. For example, you could focus on requesting demonstrations and implement a behavioural cloning method, which trains a policy network to predict the action given the state, similar to Tutorial 4. Or, you could focus on learning a model of the environment's dynamics and implement a model-based reinforcement learning algorithm, similar to Tutorial 3. Or, you could implement a hybrid method combining imitation learning and reinforcement learning, which is covered in Lecture 5. So, there is a vast space to be creative here, and everybody's solutions will be different!

To begin, I suggest implementing a basic behavioural cloning method, and implementing a basic model-based reinforcement learning method, and then comparing their performance. From here, you may then be able to improve the algorithms or their hyper-parameters, or implement some of the more advanced ideas from the lectures. And you could start by evaluating on an "easy" environment, such as one with lots of the fast, dark regions. Random seed 1707366464 is a good example of this (this can be set in *configuration.py*).

The Rules.

Whilst implementing your method in *robot.py*, you must abide by the following rules:

- *robot.py* must not import *environment.py*, for example to gain access to the true environment dynamics. When we evaluate your code, *environment.py* will actually be renamed to something else, to prevent you from doing this.
- *robot.py* must not import any other external libraries, apart from the ones specified at the start of this document, in "Setting Up".
- You must not use the GPU to train any neural networks; this must only use the CPU.
- You must not use any multi-threading.
- You must not download or copy code from elsewhere and paste this into *robot.py*, and we will be checking for plagiarism. You are welcome to read beyond the lecture

contents to discover new ideas for your implementation, but the implementation itself must be all your own code.

- In *robot-learning.py*, you must not let the money available drop below £0. This is partially handled by the code in *robot-learning.py*, but technically it would still be possible for *robot.py* to spend a very long time doing some processing right at the end, such that when the program returns to *robot-learning.py*, the money available is significantly below £0. Therefore, as can be seen in *robot-learning.py*, if the money available is less than -£1, a 10% penalty will be applied to the score you receive during testing.

If you are not sure whether the method you have implemented is "allowed", then please ask on EdStem and I will let you know.

What to Write for coursework_2.pdf

This should be a single-page document, following the template provided in "Data File", with two parts of roughly equal size. In the first part, you should write a high-level summary of your overall method, making sure that you relate your method back to topics introduced during the lectures. In the second part, you should briefly describe one experiment you ran when you were developing your method, and provide a figure showing a graph or chart that you created as part of this experiment, which you should reference in your description of the experiment. This could be similar to some of the graphs you plotted for Coursework 1. For example, this graph could show the trend between two variables, which you then used to determine an optimal hyperparameter. Or it could show the results of an experiment you conducted to compare the performance of two different implementation ideas.

Grading

Running Our Evaluation of robot.py.

When we evaluate your *robot.py* file that you submit to Scientia, we will effectively use the same code that is provided to you in the other files (*robot-learning.py, environment.py, graphics.py,* etc.), and the same values that are in *constants.py*. We will evaluate your *robot.py* on 10 different random environments, each of which will be just a random instance of the *Environment* class in the code provided. All students will be evaluated on the same 10 environments. The total score for your *robot.py* will be the average score across all 10 environments.

During testing, as can be seen in *robot-learning.py*, we will record how long it takes your robot to reach the goal (this is triggered when the distance to the goal is below a threshold). If the robot does not reach the goal, we will record the shortest distance between the robot and the goal during the testing period.

Grading for robot.py.

The quicker your robot can reach the goal during testing, or the closer it can get to the goal if it cannot reach it, the higher your score will be. Grading will be done relative to the results of two baselines that we have created. As such, your grade will be normalised based on how difficult the particular environment is. For example, some environments may be very

challenging, and it would be very difficult, or perhaps even impossible, for the robot to reach the goal, even with an excellent method. So for these environments, you could still receive a high score even if the robot could not actually reach the goal during testing. You will not be assessed on the quality of the written code, such as how well structured or commented it is; you are only being assessed on the performance of the code.

Important.

When you do your final test of your *robot.*py before submission, make sure that you run your code using exactly the same files provided on Scientia, with only modifications made to *robot.py*. You may wish to re-download the code from Scientia to make sure that any edits you made to the other files are removed. If there are any bugs and your program does not run, you may receive a reduced grade, even if these are tiny bugs, such as typos. We will not correct your own bugs or typos. We will run your code exactly as it is in your submission on Scientia, so you should also remove anything that will slow it down, e.g. print statements. In previous years, several students lost marks because they did not test their code properly before submitting, and their code caused an error before *robot-learning.py* was able to fully run the training and testing. Don't be one of those students!

Grading for coursework_2.pdf.

This document does not need to be highly detailed and elaborate, and it will be marked generously. If you can show that you have implemented something reasonable, and that you have run a sensible experiment, you can easily receive full marks. However, we are looking for implementations which generally follow the methods introduced during the lectures, and so implementations which do not use any of the methods from the lectures may not receive full marks. I anticipate that the majority of students will receive full marks for this document.

Still to Come

Over the coming days, I intend to release more information about the baselines we will be using to measure your performance against, such as example scores for a number of environments. I may also be able to set up an environment on LabTS for you to run your own evaluation using the same CPU that we will be using, and if this is possible I will be in touch about it soon. Finally, in the final lecture, we will be having a fun, informal competition, where you will be able to "race" your robot against those of other students, and further information on this is coming soon!

Any further questions?

For any other questions, please ask me on EdStem, and I promise I will respond! And I will keep everybody updated on any common issues that may arise.

I really hope that you enjoy this coursework, and I hope that it helps you to understand in more depth how robot learning algorithms are implemented, and what the important design choices are. I look forward to reading about some of your interesting methods, and to discussing some of your ideas with you in the lab sessions, and I also very much look forward to the live competition in the final lecture!

End of Coursework 2